Research Report 16/97

Specifying Frameworks and Design Patterns as Architectural Fragments

by

Jan Bosch

Department of
Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HK/R-RES—97/16—SE

Specifying Frameworks and Design Patterns as Architectural Fragments

by Jan Bosch

ISSN 1103-1581
ISRN HK/R-RES—97/16—SE

Specifying Frameworks and Design Patterns as Architectural Fragments

Jan Bosch
University of Karlskrona/Ronneby
Department of Computer Science and Business Administration
S-372 25 Ronneby, Sweden
e-mail: Jan.Bosch@ide.hk-r.se
www: http://www.ide.hk-r.se/~bosch

Abstract
Object-oriented software architectures, such as design patterns and frameworks, have found extensive use in software industry. As a design technique, these architectural structures have proven themselves very valuable. Their implementation in traditional object-oriented programming languages, however, is often difficult, since these languages provide no support for the explicit specification of software architectures. In this paper, we develop the notion of *architectural fragments*, i.e. reusable architectural specifications that describe a design pattern or a framework architecture. An architectural fragment describes the structure of an architecture in terms of its components (roles), the architecture-specific behaviour of the components and the architecture-specific interaction between the components. Fragments can be composed with each other and with reusable components. To illustrate the use of fragments, we describe the role and architecture language constructs in the layered object model (**LayOM**) and present examples.

1 Introduction

Reusable object-oriented software architectural structures such as frameworks and design patterns have proven highly valuable assets for software development. During design, design patterns are extensively used by software engineers for the actual design process as well as for communicating a design to others. The implementation of design patterns, however, has received considerably less attention and most consider the implementation of design patterns in a conventional OO programming language to be a time-consuming but necessary activity [Budinsky et al. 96]. For object-oriented frameworks, one can identify an similar situation, i.e. the software industry makes extensive use of object-oriented frameworks and frameworks are generally recognised to reduce the required development effort for applications. However, the use of frameworks during the implementation of applications has received less attention, even though the process might be far from trivial.

One of the underlying causes for these problems, we believe, is due to the fact that conventional object-oriented languages provide no support for representing architectural structures as first-class entities. A design pattern, being used as a conceptual entity during design, is implemented as a collection of code pieces embedded in two or more classes. An object-oriented framework, generally treated as an identifiable entity during design, easily loses this structure during implementation due to the large amount of classes and the inability to distinguish between relevant, i.e. entities part of the framework architecture, and irrelevant classes, i.e. entities providing reusable implementations for architecture components.

The reason why no support for describing architectural structures is available in conventional languages is that the paradigm on which these languages are based requires that the behaviour of a single software component is described as one, complete specification. Architectural structures, on the other hand, describe the part of the behaviour of a group of components related to the architecture.

Inheritance in object-oriented programming languages addresses this up to some extent. However, as identified in the work on mixins [Bracha & Cook 90], inheritance is unable to deal with the composition of class behaviour from different sources if these behaviours are not orthogonal. Mixins only solve the problem of behaviour composition for individual classes and provide no support for architectural structures that encompass multiple components.

Since no language support is available, the implementation of architectural structures suffers from several problems. In [Bosch 96], we identified that the implementation of design patterns may suffer from problems related to traceability, self problem, reusability and implementation overhead. In [Bosch et al. 97] and [Mattsson & Bosch 97], we identified that framework implementation, among others, may suffer from problems related to implicit architecture, cross-framework dependencies, framework instantiation, legacy components and framework composition.

In this paper, we propose to describe reusable parts of application architectures, e.g. design patterns and framework architectures, as *architectural fragments.* An application architecture is, in this approach, composed from multiple architectural fragments, requiring composability of these fragments in order to construct an application. The problems and requirements of specifying architectural fragments are discussed and an integrated language model is proposed for describing architectural fragments in addition to traditional object-components such as classes and objects. The composition of an application from one or more architectural fragments and classes is achieved using a composition technique called *superimposition* [Bosch 97].

The contribution of this paper, we believe, is that an approach to explicitly specifying software architectures, i.e. architectural fragments, is proposed that provides support for reusable, first-class specification of object-oriented architectures. In addition, the approach is concretised using a language model that is suitable for specifying architectural fragments and reusable components and their instantiations in applications.

The remainder of this paper is organised as follows. In the next section, the problems of architectural definition languages and programming languages for specifying architectural fragments are described. Section 4 introduces the conceptual notion of architectural fragments. Section 5 introduces the layered object model that, in section 6, is extended with functionality for specifying architectural fragments. In section 7, the approach proposed in this paper is compared to related work and the paper is concluded in section 8.

2 Example

To illustrate the ideas presented in this paper, an object-oriented framework for measurements systems will be used. The framework has been designed in cooperation with EC-Gruppen, a swedish company developing, among others, embedded systems. Measurement systems are a class of systems used to measure the relevant aspects of a process or product. These systems are different from the, better known, process control systems in that the measured values are not directly, i.e. as part of the same system, used to control the production process that creates the product or process that is measured. A measurement system is used for quality control on produced products that can then be used to separate acceptable from unacceptable products or to categorise the products in quality categories. In some systems, the results from the measurement are stored in case in the future the need arises to refer to this information, e.g. if customers complain about products that passed the measurement system.

A typical measurement cycle in a measurement system starts with a trigger indicating that a product, or measurement item, is entering the system. The first step after the trigger is the *data-collection* phase by the sensors. The sensors measure the various relevant variables of the measurement item. The second step is the *analysis* phase during which the data from the sensors is collected in one representation and transformed until it has the form in which it can be compared to the ideal values. Based on this comparison, certain discrepancies can be deduced which, in turn, lead a classification of the measurement item. In the third step, *actuation*, the classification of the measurement item is used to perform the actions that are associated with the classification of the measurement item. Example actions may be to reject the item, causing the actuators to remove the item from the conveyer belt and put it in a separate store, or to print the classification on the item so that it can be automatically recognised at a later stage. One of the requirements on the analysis phase is that the way the transformation takes place, the characteristics based on which the item is classified and the actions associated with each classification should be flexible and easily adaptable, both during system construction, but also, up to some extent, during the actual system operation.

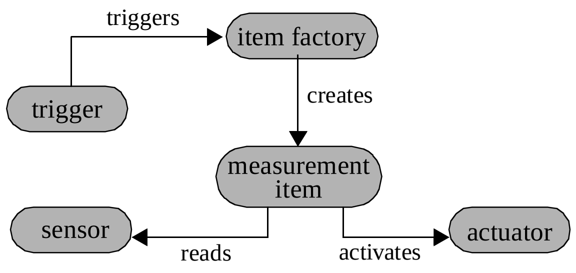

Figure 1. Measurement system architecture

In figure 1, the architecture of the measurement system is shown. A measurement system consists of one or more sensors that measure aspects of a measurement item, one or more actuators causing some effect in the real world based on the measured values and the control, a trigger identifying the entrance of a physical measurement item into the system, the item factory that generates measurement item in response to the trigger and a measurement item object that represents its real-world counterpart in the system.

The measurement system architecture makes use of a number of design patterns, among others the observer, abstract factory and strategy patterns. In this paper, the framework architecture and the observer pattern will be used as examples.

3 Motivation

Design patterns and object-oriented frameworks are generally identified as powerful techniques during design, but their implementation has received considerably less attention. However, their implementation using traditional object-oriented languages suffers from a number of problems. In [Bosch 96] we identified the following problems associated with the implementation of design patterns:

- **Traceability**: A design pattern can generally not be traced in the implementation since the programming language does not support a corresponding concept. This problem has also been identified by [Soukup 95].
- **Self problem**: The implementation of several design patterns requires forwarding of messages. Once the message is forwarded, the reference to the object originally receiving the message is no longer available. The problem is known as the *self problem* [Lieberman 86].
- **Reusability**: Since design patterns generally specify part of the behaviour a group of objects, patterns have no first class representation at the implementation level. Hence, the implementation of a design pattern cannot be reused.
- **Implementation overhead**: Implementation of a design pattern often requires several methods with only trivial behaviour, e.g. forwarding a message to another object or method

In [Bosch et al. 97, Mattsson & Bosch 97] the problems associated with the implementation of object-oriented frameworks that we identified are described. The most relevant ones are:

- **Implicit architecture**: When using a framework, the first step is to understand its architecture. Since the implementation of the framework is often dominated by implementation classes that provide reuse at the implementation level, the architecture of the framework disappears in the implementation details and can be hard to identify by the software engineer.
- **Cross-framework dependencies**: A typical situation in framework-based application development is when selecting a class in one class hierarchy limits the selection of classes to a subset in another class hierarchy. Such dependencies are often implicit and identifying them requires considerable understanding of the internal workings of the framework.
- **Framework instantiation**: Due to the implicit architecture and cross-framework dependencies, but also by the sheer size of most frameworks and the lack of a clear, well-defined framework interface, instantiating a framework requires considerably understanding of the framework internals.
- **Legacy components**: Legacy components can generally not be used in combination with frameworks without considerable adaptation, due to typing conflicts and the *cohesive behaviour* required from framework classes.
- **Framework composition**: Frameworks are generally designed to form the reused part of closed applications, rather than to be components in a larger system. This leads to composition problems, such as

3.1 Architectural description languages

The specification of architectures has been studied been studied by the software architecture community, in particular by those working on *architectural description languages* (ADLs). One can identify three categories of ADL approaches in this domain, i.e. general purpose ADLs, e.g. Rapide [Luckham et al. 95] and Darwin [Magee et al. 95], domain specific ADLs, e.g. MetaH [Binns et al. 94], and generative ADL approaches, e.g. the Aesop system [Garlan et al. 94], providing a general purpose basis, but the ability to be extended with domain-specific language constructs for specifying architectures in a particular domain.

The approach on which ADLs are based assumes a dichotomy between the architecture and the components. The architecture specifies *what* components play a role in the architecture and *how* these components are connected to each other. The components, on the other hand, are supposed to contain both the domain functionality and the *archi-*

tecture-specific behaviour associated with the architecture. For example, in an architecture based on the pipe&filter style, the components should not only contain the domain functionality, but should be able to act as a filter.

The position taken in this paper is that the architecture specification should, in addition to specifying what components the system consists of and how they are connected, also specify the architecture-specific behaviour of the components. The components should only contain domain-specific behaviour.

Although architecture description languages have explicitly been defined for the specification of software architectures, these approaches lack some aspects that we believe to be important for the use of architecture specifications in combination with existing software. Below, these issues are discussed in more detail.

- **Architecture-specific behaviour of roles**: An architecture specification generally specifies a number of roles that have to be played by components that are selected when instantiating the architecture. Roles are often required to exhibit behaviour that is specific to the architecture. ADLs do not provide support specifying architecture-specific role behaviour as part of the architecture, but require the component selected for the role to contain this behaviour. This is problematic since it reduces the usefulness of specifying an architecture independent of the components that play a role in the architecture and, secondly, since it reduces the reusability of the components that are required to contain architecture-specific behaviour, potentially complicating the use of the components in other architectures.
- **Reusability of architectures**: Most ADL designers assume that the ADL is used to describe the top-level design of applications in a particular domain, i.e. a DSSA, or the architecture of a system. The resulting architecture description itself is reusable, but the approach does not allow for reusable specification and instantiation of architectural styles or patterns.
- **Composition of architectures**: In general, most software engineers will associate the term “architecture” to the top level structure of a software system. Consequently, existing ADLs generally do not provide means to compose two or more architectural specifications. Due to this, the software engineer is forced to define an application architecture from scratch rather than to compose it from existing architecture specifications.

3.2 Requirements

From the discussion in the above sections, it can be concluded that object-oriented software architectures, such as frameworks and design patterns, cannot be implemented without problems either using a traditional object-oriented language or using an ADL. Based on the identified problems, the requirements that should be fulfilled by a description approach can be identified:

- **First-class representation**: ADLs were developed in recognition of the importance of specifying an architecture explicitly. The first-class representation of an architecture is a prerequisite for reuse and composition.
- **Architecture-specific behaviour**: The behaviour of components that is specific for the architecture should be specified as part of the architecture description. This increases the reusability of components and the usefulness of specifying architectures first-class.
- **Configurable**: An architecture description should both be *instantiatable* and *configurable*. An architecture is instantiated for a system and configured for the particularities of the system.
- **One language model**: The ADL community explicitly chose to define a separate language for specifying architectures, rather than extending an existing programming language. We believe that it is possible and preferable to define a unified language model for reasons of complexity and inter-operability.

4 Architectural Fragments

Conventional object-oriented languages allow one to describe the behaviour of one class as a single monolithic description, even though the class may inherit from other classes. Object-oriented software architectures, e.g. design patterns and object-oriented frameworks, require support for describing part of the behaviour of a group of components as a first-class entity. The entity describes the component behaviour related to the pattern or framework and the interaction between the components. However, other behaviour that the component should exhibit, e.g. application domain functionality, should not be described, but composed at a later stage.

The solution to specifying object-oriented architectures proposed in this paper is the notion of *architectural fragments*. An architectural fragment describes, for the set of components that play a role in the fragment, that part of the behaviour that is specific for the architecture. An architectural fragment consists of a set of roles and initialisation code that is executed when the architectural fragment is instantiated. A role is specified similar to a class, but it provides additional expressiveness for defining a required *interface* that has to be fulfilled by the class that is composed with the role and for configuring an instantiated role to match the interface of a class. The interface declares those operations that do not belong to the architecture behaviour but that need to be referred to by the roles in the architecture.

Different from traditional ADLs, an architectural fragment defines a partial implementation of the roles. When selecting a class to play a role in an instantiated architecture, the class behaviour and the role behaviour are composed. This sometimes requires that the role behaviour overrides or extends class behaviour. This is achieved using *superimposition*. Since superimposition is transparent, the component can generally be used as before by components outside the architecture. Although able to override the component's behaviour, the superimposed role behaviour often adapts and extends the behaviour of the class so that also the roles in the architecture can make use of the class.

In figure 2, the principle approach of architectural fragments is presented graphically. The example consists of two architectural fragments and three classes. The first architectural fragment consists of three roles whereas the other contains two roles. The three classes contain application domain behaviour, but lack behaviour for playing a role in the software architectures, e.g. design patterns or frameworks, defined by the architectural fragments. In the application, the two architectural fragments are superimposed on the domain classes. For class c_1 this causes the role behaviour of $r_{b,1}$ to be superimposed on c_1. The role behaviour from $r_{a,1}$ is superimposed on the resulting component, resulting in the presented application entity.

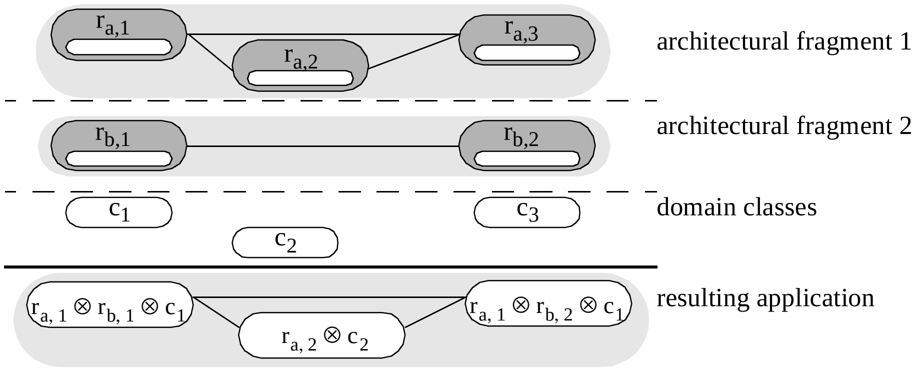

Figure 2. Illustrating the use of architectural fragments

In the remainder of this section, the introduced concepts are discussed in more detail. In the next section, the notion of *role* is discussed. Subsequently, in section 4.2, the technique that we use for composing roles with other roles and classes, i.e. superimposition, is introduced. Finally, the notion of architectural fragments is discussed in section 4.3.

4.1 Roles

Different from a traditional class description that defines the complete behaviour of a component, a role describes only that part of a component that is specific for a particular aspect of its behaviour. For instance, a role may describe the behaviour specific for a class playing a part in a design pattern. As mentioned, a role is specified similar to a class, with the difference that a required interface can be specified and that an instantiated role can be configured for composition with a particular class.

The specification of a role can differ rather widely. The minimal specification of a role is to just specify an interface defining what methods a class playing the role should provide. The maximal specification of a role is a complete definition of a component that does not need to be composed with a class in the application. That is, the role specifies all the behaviour required for its correct operation by itself and is not supposed to be composed with a component.

Figure 3 illustrates the composition of a role with a component. The component *C* consists of an instance variable *v1* and two methods *m1* and *m2*, whereas the role *R* consists of an instance variable *v2* and two methods *m3* and *m4*. The result of the composition is shown in the figure. Since the role composition is transparent, clients of the component or the role can still invoke methods of the component, i.e. *m1* or *m2*.

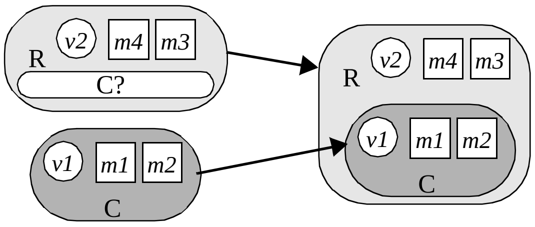

Figure 3. Composing a role and a component

Instantiated roles intercept messages sent to the object. A message is intercepted by the component if it contains, perhaps after the message has been altered (see section 5 for how to modify messages), a method that matches with the invoked message. The method in the role will execute its code and can then perform one of two actions, i.e. send a reply to the sender of the message by using a normal return statement or allow the message to proceed to the part of the component that is encapsulated by the role, by using the proceed statement. In addition to intercepting normal messages, a role can also intercept return messages both sent back to or returned by the component. Intercepting return messages allows a role to add additional behaviour after the original behaviour was executed and not only before the message is handled in the component core. Below, an example role containing each type of adding and overriding component behaviour:

```
Role R
    methods
        overiding_method(args, ...)
            begin
                    ... new response behaviour ...
                    return return_obj;
            end;
        pre_extended_method(args, ...)
            begin
                    ... additional behaviour ...
                    proceed;
            end;
        return from post_extended_method(args, ...)
            begin
                    ... additional behaviour ...
                    proceed;
            end;
end;
```

To give an example of using the role concept, an example from the measurement system framework is used. The system contains, among others, sensors that measure relevant properties of a measurement item. The description of the sensor is shown below in pseudo-code:

```
class Sensor
    variables
        value : MeasurementValue;
    methods
        read returns MeasurementValue
            begin ... end;
        refresh returns Boolean
            begin ... end;
        calibrate returns Boolean
            begin ... end;
end;
```

The sensor is, obviously, not an ‘island’ and the data it receives from its real-world counterpart needs to be communicated to other entities in the system. To achieve that, at least three solutions exist:

- **Observer pattern**: One can implement the *Observer* pattern [Gamma et al. 94] as part of the component behaviour. Any dependent object will be notified whenever the sensor value changes and can subsequently request the new sensor value.

- **Blackboard**: If the *blackboard* architectural style chosen for the system, individual components submit their results to the blackboard. Thus, when the sensor is updated, it sends a tuple <sensor-id, value> to the blackboard. Other components can access the latest sensor value by requesting it from the blackboard.
- **Data pushing**: Instead of letting dependent objects request data from the sensor, the data pushing approach ‘pumps’ data further in the system as soon as it becomes available. In this case, the sensor has references to the components interested in sensor data and sends updated values to these components.

Each of these approaches has its advantages and disadvantages and which one is selected depends on the type of system and the preference of the software engineer. In our cooperation with software industry, we have seen examples of each approach.

The reusability problem created by this example is that in traditional object-oriented programming, the developer of the sensor component is forced to select one type of communication and embed it in the sensor component. The component can now only be reused for those systems where the same type of communication approach is selected.

Below the role implementations for the approaches to data communication are presented.

```
Role Subject
  variables
    dependents : Collection;
  methods
    addDep(d : Object) returns Boolean
      begin ... end;
    removeDep(d : Object) returns Boolean
      begin ... end;
    return from change()
      begin
        for each d in dependents
          d.changed();
      end;
end;
```

```
Role BBupdater
  methods
    return from change()
      begin
        bb.newTuple(self.id(), self.data());
      end;
end;
```

```
Role DataPusher
  variables
    sinks : Collection;
  methods
    initialize( dependents: Collection)
      begin sinks := dependents; end;
    return from change()
      begin
        for each s in sinks
          s.update(self.data());
        end;
end;
```

When constructing the application, the software engineer can reuse the sensor class and select one of the roles for data communication between the sensor and the rest of the system. When instantiating a particular role, one has to configure the role in order to connect the role behaviour to the correct class behaviour. Below, three example instantiations of the sensor class are shown using the different roles. The examples make use of layers for adapting the selectors in the messages sent to or from the components. Layers are discussed in section 5.

```
application SimpleMeasSystem1 begin
  components
    s : sensor with roles
      Subject with layers
        a : Adapter(accept update as change);
      end;
    end;
    ...
end;
```

```
application SimpleMeasSystem2 begin
  components
    s : sensor with roles
      BBupdater with layers
        o : Adapter(accept update as change);
      end;
    end;
    ...
end;
```

```
application SimpleMeasSystem3 begin
  components
    s : sensor with roles
      DataPusher with layers
        o : Adapter(accept update as change);
      end;
    end;
    ...
end;
```

The application description contain an instantiation of the sensor class and of a role. The role defines how the value read by the sensor component is communicated to other parts of the system. However, an instantiated role needs to be configured for the particular object is imposed upon. The Observable role uses an Observer layer to be notified when the sensor is updated, whereas the BBupdater and DataPusher roles make use of an adapter layer to connect the method names used by the sensor object to those used by the role.

Due to limited space, only the composition of a single role with a class is presented here. Section 6 discusses an example of composing multiple roles with a class.

4.2 Superimposition

Superimposition [Bosch 97] was defined in response to the difficulties of traditional component adaptation and composition techniques, such as inheritance, aggregation and wrapping, with respect to the composition of role and class behaviour. Certain types of role functionality need to be integrated with the component's behaviour that are orthogonal to its structural parts and may affect e.g. multiple methods. The software engineer needs to *superimpose* certain behaviour on a component in such a way that the complete functionality of the component is affected. The notion of superimposition in computing systems has been identified before, but not in object-oriented or component-based systems. For example, [Bouge & Francez 88] define and use superimposition in the context of CSP. They define the superimposition *R* of *P* over *Q* as the additional superimposed control *P* over the basic algorithm *Q*. Analogously, we define object superimposition *S* of *B* over *O* as the additional overriding behaviour *B* over the behaviour of component *O*.

To define superimposition in more detail, some parts of a more precise object model specification is presented. An *object* o is defined as $o = (I, M, S, P)$ where *I* indicates the interface of the object, *M* the set of methods, *S* the state space formed by the instance variables and *P* the mapping from the interface to the methods. The interface of the object is the set of message selectors that the object can respond to. For a basic object, $I = \{m \in M | \mathrm{selector}(m)\}$ where $\mathrm{selector}(m)$ is a function that returns the selector using which method *m* can be invoked. The mapping *P* is defined as $P{:}I \to M \vee \mathrm{reject}$, i.e. each interface element is either mapped to a method in the method set or rejected. Also, we define the behaviour *B* of an object as $B = \{m \in M | b_m \Rightarrow S \to S\}$. The behaviour b_m of a method *m* is defined as the state change of the object caused by an execution of *m*. Finally, the notion of a message is defined as $e = (n, r, l)$, where *n* is the sender of the message, *r* is the receiver and *l* is the selector. We do not incorporate the message arguments in the model for reasons of simplicity, neither do we incorporate state changes at other objects due to messages sent by a method executed at o.

An object can have a set of roles *G* associated with it; $G = \{g_1, \ldots, g_p\}, p \geq 0$. A role *g* is defined as $g = (I, M, S, P, R)$, where *I* and *S* are similar to the definition for objects and *R* is the interface required from the object the role is composed with. $M = (M^o, M^e, M^r)$, where M^o indicates the set of overriding methods, M^e the set of extending methods and M^r the set of return message extending messages. Assuming the role is composed with an object $o = (I_o, M_o, S_o, P_o)$, the mapping function *P* is defined as

$$P = \begin{cases} i \to m^o & m^o \in M^o \wedge \mathrm{name}(m^o) = i \\ i \to m^e;m & m^e \in M^e \wedge m \in M_o \wedge \mathrm{name}(m^e) = \mathrm{name}(m) = i \\ i \to m;m^r & m^r \in M^r \wedge m \in M_o \wedge \mathrm{name}(m^e) = \mathrm{name}(m) = i \\ i \to m & m \in M_o \wedge \mathrm{name}(m) = i \\ i \to \mathrm{reject} & i \notin I_g \wedge i \notin I_o \end{cases}$$

A role can be composed with an object *o*. The composition $o' = g \otimes o$ leads to a new object o' that is an adaptation of *o* since aspects of it may be changed, i.e. $g \otimes o = (I_g \cup I_o, M_g \cup M_o, S_g \cup S_o, P_g)$, provided $P \subseteq I_o$. Since the interface of the resulting object o' is indistinguishable from a basic object, clients treat both types of objects equivalent. An object superimposed with multiple roles can be defined as $g_1 \otimes \ldots \otimes g_p \otimes o, p \geq 1$.

4.3 Architectural Fragments

An architectural fragment (or fragment) describes two or more roles and their interaction. Using superimposition, an architectural fragment can be composed with other fragments and with classes. Fragments support the specification of object-oriented software architectures as first-class entities at the implementation level. Since architectures are treated as entities at the design level, corresponding concepts at the implementation level improves traceability and conceptual integrity of the implementation and the design.

To illustrate the use of fragments, the *Observer* pattern is used as an example. The fragment consists of two roles, i.e. the subject and the observer. The subject role maintains a collection of dependent objects, i.e. observers, and contains methods for adding and removing dependents. In addition, it contains an observable method, which represents those methods in the class composed with the model role that need to be observed. The observedMethod notifies the dependents and subsequently forwards the message to the class composed with the role using the proceed statement. The observer role only defines an interface required from the class composed with the role, i.e. the update method. This method is required since the subject role depends on it. However, when instantiating an architecture, the roles can be configured to match the class that they are composed with. For instance, the update method can be renamed into another method name provided by the class.

```
architecture Observer
    roles
        subject
            variables
                dependents : Collection;
            methods
                addDependent(newDependent : Object)
                    begin ... end;
                removeDependent(aDependent : Object)
                    begin ... end;
                observedMethod()
                begin
                        for each d in dependents do d.update(self);
                        proceed;
                end;
        end;
        observer
            interface
                update(Object);
        end;
    initial
            begin
                subject.addDependent(observer);
            end;
end; // Observer
```

The *Observer* architecture can be instantiated in an application and composed with domain classes. An example is shown below. Two objects, i.e. instances of the sensor class and measurement item class, respectively, and an instance of the Observer architecture are composed. The instantiation of the architecture has to be configured, since there are mismatches in expected and supported method names. The configuring is performed using *Adapter* layers (see next section) that convert message selectors. The sensor object is composed with the subject role and the refresh message selector is changed for the subject role into observedMethod. Note that this adaptation is only valid for the role and not for the encapsulated object. Thus, once the proceed statement is executed, the encapsulated object will receive a refresh message. The measurement item object is adapted to accept the update message sent by the subject role to its observers as a getValue message, which is one of the methods on its interface. In this case, the encapsulated object does receive the adapted message selector, i.e. getValue, since the message is not intercepted by the role. In section 6, more extensive examples of using architectural fragments are given.

```
application ObserverExample begin
    architectures
        anObsArch : Observer where
            s is-a subject with layers
                o : Adapter(accept refresh as observedMethod);
            end;
            mi is-a observer with layers
                a : Adapter( accept update as getValue);
            end;
        end;
    components
        s : Sensor;
        mi : MeasurementItem;
end; // ObserverExample
```

5 Layered Object Model

The concepts of architectural fragments, roles and superimposition require a concrete language model to illustrate their use and for experimenting with concrete examples. For this purpose, an extension to the layered object model

(**LayOM**), our experimental research language is discussed. **LayOM** has been applied to several problems associated with the traditional object-oriented paradigm, such as representation of inter-object relations, design patterns, abstract object state, acquaintance handling and component adaptation. In this paper, a solution to the specification of architectural fragments as reusable first-class entities is discussed.

The layered object model is an extended object model, i.e. it defines in addition to the traditional object model components, additional components such as layers, states and acquaintance categories. In figure 4, an example **LayOM** object is presented. The layers encapsulate the object, so that messages send to or by the object have to pass the layers. Each layer, when it intercepts a message, evaluates the contents to determine the appropriate course of action. Layer classes have, among others, been defined for the representation of inter-object relations, design patterns, component adaptation and acquaintance handling.

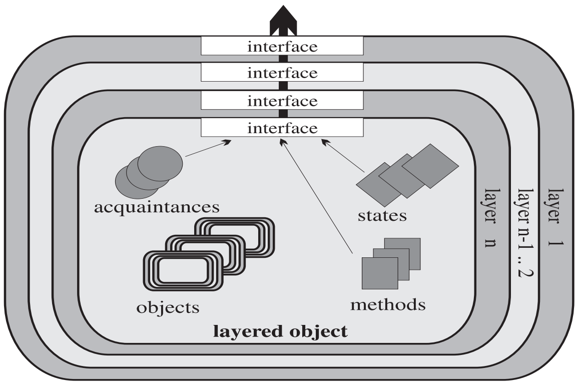

Figure 4. The layered object model

A **LayOM** object contains, as any object model, instance variables and methods. The semantics of these components is very similar to the conventional object model. The only difference is that instance variables, as these are normal objects, can have encapsulating layers adding functionality to the instance variable. A *state* in **LayOM** is an abstraction of the internal state of the object. In **LayOM**, the internal state of an object is referred to as the *concrete state*. Based on the object's concrete state, the software engineer can define an externally visible abstraction of the concrete state, referred to as the *abstract* state of an object.

An *acquaintance* defines a set of objects that are treated similar by the object by describing the discriminating characteristics of a subset of the external objects that should be treated equally by the class. Among others, behavioural layer types use acquaintances to determine whether the sender of a message is a member of an acquaintance category. If the sender is a member, the message is subject to the semantics of the specification of the behavioural layer type instance.

A layer encapsulates the object and intercepts messages. It can perform various kinds of behaviour, either in response to a message or pro-actively. To illustrate the behaviour of layers, figure 5 presents the RestrictClient layer type, which represents a behavioural relation. The layer intercepts messages sent to the object. If the message is sent by an object that is not classified as a member of the acquaintance category indicated by the layer specification, the message will just be passed on to the next layer. Otherwise, the selector of the message is matched with the identifier list in the layer specification. Depending on the layer specification, the message may be passed on to the next layer (see (a) in figure 5). If the message is not passed on, the message may be stored in the message queue (see (b) in figure 5). Otherwise, the message is rejected and an error message is sent to the sender object (see (c) in figure 5). Note that the conceptual execution model described here does not necessarily represent the actual implementation. Different translation approaches have been designed that generally favour performance over required implementation effort or visa versa.

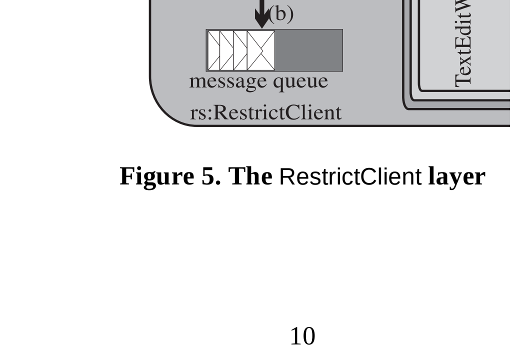

Figure 5. The RestrictClient **layer**

Next to being an extended object model, the layered object model also is an *extensible* object model, i.e. the object model can be extended by the software engineer with new components. **LayOM** can, for example, be extended with new layer types, but also with new structural components, such as *events*. The notion of extensibility, which is a core feature of the object-oriented paradigm, has been applied to the object model itself. Object model extensibility may seem useful in theory, but in order to apply it in practice it requires extensibility of the translator or compiler associated with the language. In the case of **LayOM**, classes and applications are translated into C++. The generated classes can be combined with existing, hand-written C++ code to form an executable. The **LayOM** compiler is based on *delegating compiler objects*, a concept that facilitates modularisation and reuse of compiler specifications and extensibility of the resulting compiler. We are currently working on integrating architectural fragments and roles into the language implementation.

5.1 Role Construct

In section 4.1, the conceptual notion of a role was introduced. In **LayOM**, the concept has been implemented using the Role language construct. A role provides a reusable specification of the object behaviour for one aspect particular aspect, e.g. playing one of the characters in a design pattern or handling communication with other objects. A role is specified very similar to a **LayOM** class, but it has additional expressiveness for defining the interface required of the class the role can superimposed with and for configuring an instantiated role to match the interface of the concrete object. Below, the most relevant parts of the syntax of the role are shown. A role specification consists of six main elements, i.e. variables, methods, acquaintances, states, layers and an interface. Each element may be omitted and, if present, may be defined in any order. The acquaintances part declares objects that an instantiated role requires to have a binding to in order to function correctly. Layers of type Acquaintance are used to bind the acquaintance name used inside the role to an external object. The interface declares the methods that should be supported by a class that is composed with the role.

```
Role R
    variables
        x : <variable_type>;
        ...
    methods
        <method-name> <arguments> ...
        ...
    acquaintances
        acq_1;
        ...
    states
        state_1 ...
        ...
    layers
        l : <layer_type> ...
        ...
    interface
        <method-name>;
        ...
end;
```

5.2 Architecture Construct

The architectural fragment concept was introduced in section 4.3. This concept has been implemented using the architecture language construct in **LayOM**. The most relevant parts of the architecture syntax are shown below. An architecture consists of one or more roles and a piece of code executed when the fragment is instantiated. The roles can either be specified as part of the architecture specification, e.g. role-1, or specified separately and included in the architecture, e.g. role-2. If a role is included, it may need to be adapted, e.g. role-3. Adapting a role is performed using layers.

```
architecture A
    roles
        role-1
            ..
        end;
        role-2;
        role-3 with layers
            l : <layer-type> ...;
            ...
        end;
    initial
            begin
                ... initialisation code for the architectural fragment ...
            end;
end; // A
```

6 Illustrating Architectural Fragments

In section 2, the domain of measurement systems as well as a simplified software architecture were introduced. The measurement system architecture consists of five entity types. Of these types, multiple instances of the sensor and actuator type may appear. Below, the code for describing the domain specific software architecture for measurement systems is shown. The *sensor* and *actuator* roles are only specified by their respective required interfaces. The *trigger* role consists of a trigger method and an acquaintance relation to the item factory. The *item-factory* role is actually a complete object specification and no class is required for this role when the architecture is instantiated. The *measurement item* role is defined by a required interface and acquaintance relations to the sensor and actuator roles.

```
architecture MeasurementSystemArchitecture
    roles
        sensor
            interface
                read
        end;
        actuator
            interface
                activate
        end;
        trigger
            acquaintances
                item-factory;
            methods
                trigger
                    begin item-factory.trigger; proceed; end;
        end;
        item-factory
            layers
                prototype-item : PartOf(measurement-item);
            methods
                trigger
                    begin
                        measurement-item mi;

                        mi := prototypeItem.copy;
                        if (inCalibration) then mi.calibrate; end;
                        mi.start;
                    end;
                calibrate(measurement-item mi)
                    begin prototype-item.become(mi); end;
        end;
        measurement-item
            acquaintances
                sensor;
                actuator;
            interface
                start;
        end;
end; // MeasurementSystemArchitecture
```

6.1 Instantiating the Measurement System Architecture

To illustrate the use of architectural fragments, the instantiation of the measurement system architecture is exemplified using a very simple measurement system consisting of one sensor, one trigger and one actuator. Several details with respect to calibration of the system, update strategies for the sensor and measurement and actuation strategies for the measurement item are omitted from the discussion. The system is build for a ceramics factory and weighs products. If the product’s weight is between correct boundaries, the product is assumed to be acceptable. If it is outside the boundaries, there is generally something wrong with it and it should be removed from the production output. Since the product is ceramic, removing it from the conveyer belt is done easiest by a crushing device that breaks the product into small parts and removes these parts from the belt. The hardware of the system consists of a weight sensor, a light contact trigger and a crushing device. Corresponding software entities have been defined and a measurement item component specific for the ceramic pots. Below, the instantiation code for the application is shown.

```
application SimpleMeasurementSystem-example begin
    architectures
        aMS : MeasurementSystemArchitecture where
            ws is-a sensor with layers
                a : Adapter(accept value as getWeight);
            end;
            tr is-a trigger;
            ca is-a actuator;
            cpi is-a measurement-item with layers
                sensor : Acquaintance(ws);
                actuator : Acquaintance(ca);
        end;
    components
        ws : WeightSensor;
        tr : LightContactTrigger
        ca : Crusher;
        cpi : CeramicPotItem;
end; // SimpleMeasurementSystem-example
```

The application shown above instantiates the measurement system architecture and composes four components with the instantiated architecture. Since the item factory role is fully specified in the architecture, it does not need to be composed with a component for instantiation. Since the components do not have to be adapted in any major ways, the specification of the instantiation is rather brief. Graphically, the example application can be viewed as shown in figure 6.

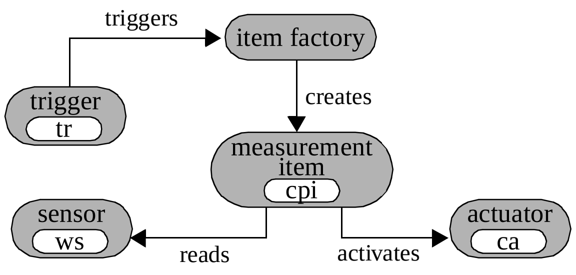

Figure 6. Measurement system application

6.2 Composition of Architectural Fragments

Multiple architectural fragments can be used for constructing an application. As a second example, a measurement system is used that inspects beer cans using a camera to verify that each can is clean. The system is located before the beer cans are filled and dirty cans are to be removed from the conveyer belt. In addition to the measurement system architecture, a user interface component is present that should present activation of the trigger and the value read by the sensor. Since beer cans enter the system at irregular times, the UI component is unable to poll. Instead, the trigger and the sensor component should notify the UI component whenever a relevant state change took place. This is a typical instance of the *Observer* pattern, but trigger and sensor roles nor the classes intended to play the role have been prepared for acting as a subject. Instead of editing the source code of these components, two instances of the Observer fragment presented in section 4.3 are used. The code of the system instantiation is shown below.

```
application SecondMeasurementSystem-example begin
    architectures
        aMS : MeasurementSystemArchitecture where
            c is-a sensor with layers
                a : Adapter(accept value as getFrame);
            end;
            tr is-a trigger;
            l is-a actuator;
```

```
                bci **is-a** measurement-item **with layers**
                    sensor : Acquaintance(c);
                    actuator : Acquaintance(l);
                **end**;
            **end**;
            o1 : Observer **where**
                tr **is-a** subject **with layers**
                    a: Adapter(**accept** trigger **as** observedMethod);
                **end**;
                ui **is-a** observer,
            **end**;
            o2: Observer **where**
                c **is-a** subject **with layers**
                    a: Adapter(**accept** getFrame **as** observedMethod);
                **end**;
                ui **is-a** observer;
            **end**;
        **components**
            c : Camera;
            tr : LightContactTrigger
            l : Lever;
            bci : BeerCanItem;
            ui : UserInterface;
**end**; // SecondMeasurementSystem-example
```

In figure 7, the specification of the application is presented graphically. The two instances of the Observer fragment are superimposed on the object implementing the domain behaviour. The resulting object is superimposed with the role behaviour for the measurement system. If the ordering of architectural fragments would have been different in the specification would the order of superimposed roles have been different also.

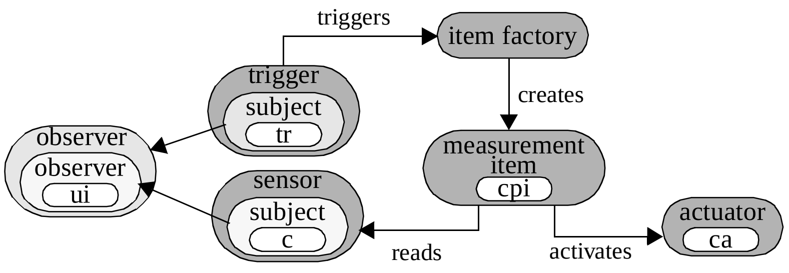

Figure 7. Composition of architectural fragments

7 Related Work

In this paper, we have investigated the problems of specifying architectural fragments as reusable first-class entities. We have proposed a solution approach that uniformly extends the object-oriented paradigm. Others have investigated these or related issues, especially the work on aspect-oriented programming [Lopez et al. 97], on implementing design patterns, mixins, meta-object protocols and on architecture description languages. Below, the relation to this work is described in more detail.

Aspect-oriented programming (AOP) is similar to the work described in this paper, but there are some major differences. [Kiczales et al. 97] suggest to describe the domain functionality in a conventional programming language and to describe the various aspects of the system using special aspect languages. At system generation time, these specifications are collected and an aspect weaver merges the domain functionality and the aspect code. The differences to the approach described in this paper are that in **LayOM** all specifications are described using one language and, instead of an aspect weaver, superimposition is used as a means to compose specifications. The **LayOM** compiler can generate code for each component independently. Another language model categorised under AOP is the composition-filters (CF) object model [Aksit et al. 93]. Although the filters in the CF model provide some forms of superimposition, the object model does not allow for modular, reusable description of architectural structures as proposed in this paper.

The implementation of design patterns has been studied by some authors. [Soukup 95] proposes to implement design patterns as C++ classes. The disadvantage of this approach is that only a few patterns can be implemented as classes.

[Budinsky et al. 96] discusses a tool for automatic code generation from design patterns. The authors also identified problems with the implementation of patterns, e.g. some designers have difficulty converting patterns into code and most designers consider implementing a chore they would rather avoid. [Reenskaug et al. 95] discusses role-based modelling of objects. Different from our approach, the authors focus on the use of roles during the design phase. During implementation, the different roles are merged (synthesis) in a conventional object-oriented class specification. In addition, no support for dealing with overriding or conflicting behaviour between roles is provided. In [Bosch 96], we describe the representation of design patterns as layer types. The model proposed in this paper extends on our earlier work in that this model also can deal with design patterns influencing the behaviour of multiple objects. In addition, the proposed model is more general in that it can also represent other architectures, such as framework architectures. [Kim & Benner 96] and [Florijn et al. 97] propose tool support for representing design patterns. Florijn's use of fragments is slightly different from ours since it includes classes, methods and associations in addition to design patterns, whereas our use of fragments only includes architectural structures. The tool support approach embodies the development of a design including design patterns and subsequent code generation in an object-oriented language. Depending on the type of tool support, patterns can be visualised in the generated code. This approach results in two abstraction levels to be used, whereas our approach integrates first-class architecture representation in the object-oriented language itself.

The work on *mixins*, e.g. [Bracha & Cook 90] and [van Limberghen & Mens 96], also identified the problems discussed in this paper but for individual classes only. Mixins provide no support for describing architecture-related behaviour for a group of classes. Roles are similar to mixins in their intention, i.e. describing an object from one perspective. Differences are that roles assume an encapsulation structure, whereas mixins are based on a (multiple) inheritance model, and that roles can be configured to match objects using layers. Meta-object protocols (MOPs) [Kiczales et al. 91, Chiba 95] provide a different, but related approach to separate different types of behaviour. A meta-object can contain role behaviour that can be activated when specified events take place in the base object. The notion of roles presented in this paper is different from MOPs in two aspects. First, the composition of MOPs is a known problem, whereas the composition of roles has a well defined and intuitive semantics. Secondly, the meta-object and the base object form no conceptual entity, whereas a component, even though it is composed from a class and one or more roles, is an entity both from a structural as well as a conceptual perspective.

An approach that could be applied to describing architectural structures is the research on *object group communication*. Most authors are concerned with representing complex communication or interaction patterns in a group of objects where each object plays some roles in the pattern. The work on *contracts* [Helm et al. 90] presents an extension to the conventional object-oriented paradigm that represents such interaction patterns and some formal aspects of such patterns. Gluons [Pintado 95] also represent interaction patterns, but contain a finite state machine to describe the interaction protocol. Whereas contracts and gluons describe restrictions on the interactions between objects, other approaches aim at specifying the actual coordinated behaviour. *Multi-methods* [Kickzales et al. 91] in CLOS define multi-polymorphic methods that are selected not only based on the receiver but on all parameters and as such can be used to define coordinated behaviour of a set of objects of predetermined types. [Schrefl et al. 96] introduce the notion of *cooperation contracts* in the context of object-oriented databases and [Järvinen et al. 90] define *joint actions*. These approaches model coordinated behaviour of a group of objects as a separate entity that can be explicitly invoked for a matching set of objects. *Abstract communication types* (ACTs) [Aksit et al. 93] is a similar approach that uses message reflection to implement interaction patterns. However, different from the three aforementioned approaches, ACTs are intended to be invoked implicitly by one of the objects in a cooperating group. The invocation of the coordinated behaviour is thus hidden from the client. All object group communication approaches are concerned with representing an interaction pattern between a group of objects as a separate entity that interacts with the involved objects. However, the interaction is explicit in that the involved objects are aware of the group entity.

As we identified in section 3, one can categorise ADLs into general purpose, domain specific and generative ADL approaches. Examples of general-purpose ADLs are Rapide [Luckham et al. 95], Unicon [Shaw et al. 95], ArTek [Terry et al. 94], Wright [Allen & Garlan 97] and RESOLVE [Ogden et al. 94]. These languages aim at specifying application architectures in terms of components and connectors between these components. The emphasis is on describing the architecture for a particular application, rather than the specification of reusable architectural patterns or fragments. Components are only specified in terms of interfaces and, since the ADLs are primarily intended for specifying application architectures, little support is provided for specifying the architecture specific behaviour of the components. For the same reason, reusability and composability of architecture specifications is not explicitly supported. Finally, to the best of our knowledge, the ADLs are specific for architecture specification and do not provide support for component specification.

Domain-specific ADLs such as MetaH [Binns et al. 94], despite their usefulness in their intended domain, do not fulfil the requirements that we have with respect to the specification of architectural fragments. Architecture-specific behaviour of roles is often specified as part of the domain-specific language model and the reusability of the architectural concepts is consequently very high in their application domain. However, composition of architecture specifications with architecture specifications from other domains is generally not easily achieved. Also, the ADLs do not provide support for the specification of components. MetaH has, for that reason, a corresponding language model, ControlH, that can be used to specify components. However, although domain-specific ADLs have addressed part of the identified problems in their particular domain, their usability is very limited outside their application domain.

The generative approaches, e.g. Aesop [Garlan et al. 94], provide a system that can generate architecture design environments for particular architectural styles and application domains. So, these approaches *generate* rather than directly provide ADLs or corresponding environments for using particular architectural styles or defining architectures in an application domain. This approach seems to support reuse of the architectural specifications and provides means to define constraints on the behaviour of components in the architectures. However, to the best of our understanding, no support is provided for component programming, architecture-specific behaviour or composition of architectures.

8 Conclusion

During recent years, software engineering research has more and more appreciated the value of software architectures and reusable architectural structures during system design. For example, the design pattern community has developed a multitude of patterns of interacting classes and objects that address particular problems. Also, the research on object-oriented frameworks and domain-specific software architectures has shown the importance of explicit architectural design knowledge.

Despite the recognised importance of architectural design expertise during design, considerably less effort has been put on providing linguistic support for specifying architectural design in an expressive, reusable and composable manner. The implementation of design patterns in conventional object-oriented languages may suffer from problems related to traceability, the self problem, reusability and implementation overhead, whereas the implementation of object-oriented frameworks may suffer from problems related to the implicit architecture, cross-framework dependencies, framework instantiation, legacy components and framework composition.

In this paper, the notion of *architectural fragments* is introduced to provide a means for representing architectural specifications, including design patterns and object-oriented frameworks, as reusable first-class entities. An architectural fragment consists of a number of roles and an initialisation method. A *role* specifies the architecture specific behaviour of the role and consists of instance variables, methods, acquaintances, layers and an interface. When instantiating an architecture, the software engineer has to select the classes that will play the roles of the architecture. The architecture-specific behaviour of the role then has to be composed with the component's behaviour. Since the architecture-specific behaviour sometimes needs to override the component's behaviour, the role needs to be superimposed on the component.

The notion of *superimposition*, i.e. the additional, overriding behaviour B over the behaviour of a component O, is not supported by traditional language models and we have, therefore, defined the layered object model (**LayOM**) that, through the use of layers, provides superimposition of various types of behaviour, such as component adaptation types and role behaviour. The layered object model has been extended with the role and architecture language construct that facilitate the specification and instantiation of architectural fragments. Both the specification of reusable architectural fragments as well as instantiations of the fragments are illustrated by several examples, such as the *Observer* design pattern and two instantiations of a framework architecture for measurement systems.

The contribution of this paper, we believe, is that it introduces the notions of *role* and *architectural fragment* and their corresponding role and architecture language construct in **LayOM** that do not suffer from the identified problems and allow for reusable first-class specification of architectural fragments. Due to the notion of *superimposition* and the **LayOM** layers that provide an implementation means, an advanced composition method is provided that facilitates the composition of roles and components.

The architecture specification language model that we develop in this paper provides the basic features necessary for specifying reusable architectural fragments that can be composed in powerful ways with suitable reusable components. However, several issues need to be investigated in more detail, such as the expressiveness required for configuring fragments and the problems that may occur when composing multiple fragments. Also, extending existing architectural specifications (‘subclassing’) and more expressive support for architecture instantiation have not been studied. Finally, we intend to experiment with the introduced concepts in more and different industrial applications. Based on the extension to the **LayOM** implementation that we are currently developing, we expect to collect more information and increase our understanding of the necessary ingredients of a language model both suitable for specifying architectures and components.

References

[Aksit et al. 93]. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, ‘Abstracting Object Interactions Using Composition-Filters,’ in *Object-based Distributed Programming*, R. Guerraoui, O. Nierstratz, M. Riveill (eds.), LNCS 791, Springer-Verlag, 1994.

[Allen & Garlan 97]. R. Allen, D. Garlan, ‘The Wright Architectural Specification Language,’ *Draft paper,* CMU, 1997.

[Binns *et al.* 94]. P. Binns, Matt Englehart, M. Jackson, S. Vestal, ‘Domain-Specific Software Architectures for Guidance, Navigation and Control,’ *Honeywell Technical Report*, 1994.

[Bracha & Cook 90]. G. Bracha, W. Cook, ‘Mixin-Based Inheritance,’ Proceedings OOPSLA/ECOOP ‘90, pp. 303-311, 1990.

[Bosch 96]. J. Bosch, ‘Design Patterns as Language Constructs,’ Accepted for publication in the *Journal of Object-Oriented Programming,* November 1996.

[Bosch et al. 97]. J. Bosch, P. Molin, M. Mattsson, PO Bengtsson, ‘Object-Oriented Frameworks - Problems and Experiences,’ *submitted,* May 1997.

[Bosch 97]. J. Bosch, ’Superimposition: A Component Adaptation Technique,’ *submitted*, October 1997.

[Bouge & Francez 88]. L. Bougé, N. Francez, ‘A Compositional Approach to Superimposition,’ *Proceedings POPL’88,* pp. 240-249, 1988.

[Budinsky *et al.* 96]. F.J. Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu, ‘Automatic code generation from design patterns,’ *IBM Systems Journal*, Vol. 35, No. 2, 1996.

[Florijn et al. 97]. G. Florijn, M. Meijers, P. van Winsen, ‘Tool Support for Object-Oriented Patterns,’ *Proceedings ECOOP’97,* pp. 472-495, 1997.

[Helm *et al.* 90]. R. Helm, I. Holland, D. Gangopadhyay, ‘Contracts: Specifying Behavioral Compositions in Object-Oriented Systems,’ *Proceedings OOPSLA’90*, pp. 169-180, 1990.

[Gamma *et al.* 94]. E. Gamma, R. Helm, R. Johnson, J.O. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[Garlan *et al.* 94]. D. Garlan, R. Allen, J. Ockerbloom, ‘Exploiting Style in Architectural Design Environments,’ *Proceedings of SIGSOFT ‘94 Symposium on the Foundations of Software Engineering,* December 1994.

[Goldberg & Robson 89]. A. Goldberg, D. Robson, *Smalltalk-80 - The Language,* Addison-Wesley, 1989.

[Järvinen et al. 90]. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, K. Systä, ‘Object-Oriented Specification of Reactive Systems,’ *Proceedings of the 12th International Conference on Software Engineering,* IEEE Computer Society Press, Nice, France, March 1990.

[Katz 93]. S. Katz, ‘A Superimposition Control Construct for Distributed Systems,’ *ACM Transactions of Programming Languages and Systems,* Vol. 15, No. 2, April 1993.

[Kickzales *et al.* 91]. G. Kiczales, J. des Rivières, D.G. Bobrow, *The Art of the Metaobject Protocol,* The MIT Press, 1991.

[Kiczales *et al.* 97]. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, ‘Aspect-Oriented Programming,’ *Proceedings of ECOOP’97* (invited paper), pp. 220-242, LNCS 1241, 1997.

[Kim & Benner 96]. J.J. Kim, K.M. Benner, ‘An Experience Using Design Patterns: Lessons Learned and Tool Support,’ Theory and Practice of Object Systems, Vol. 2, No. 1, pp. 61-74, 1996.

[Lieberman 86]. H. Lieberman, ‘Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems,’ *Proceedings OOPSLA ‘86,* pp. 214-223, 1986.

[van Limberghen & Mens 96]. M. van Limberghen, T. Mens, ‘Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems,’ *Object-Oriented Systems*, 3, pp. 1-30, 1996.

[Lopez et al. 97]. C. Lopez, ‘Aspect-Oriented Programming - Workshop Report,’ in *ECOOP’97 Workshop Reader*, J. Bosch & S. Mitchell (eds.), LNCS 1357 (forthcoming), Springer Verlag, 1997.

[Luckham *et al.* 95]. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, ‘Specification and Analysis of System Architecture Using Rapide,’ *IEEE Transactions on Software Engineering,* Special Issue on Software Architecture, 21(4):336-355, April 1995.

[Magee et al. 95]. J. Maggee, N. Dulay, S. Eisenbach, J. Kramer, ‘Specifying Distributed Software Architectures,’ Proceedings ESEC’95, 1995

[Mattsson & Bosch 97]. M. Mattsson, J. Bosch, ‘Framework Composition: Problems, Causes and Solutions,’ Accepted for *TOOLS USA’97*, 1997.

[Ogden *et al.* 94]. W.F. Ogden, M. Sitaraman, B.W. Weide, S.H. Zweben, ‘Part I: The RESOLVE Framework and Discipline - A Research Synopsis,’ *Software Engineering Notes* 19, 4, pp. 23-28, October 1994.

[Pintado 95]. X. Pintado, ‘Gluons and the Cooperation between Software Components,’ in [Nierstrasz & Tsichritzis 95], pp. 321-349, 1995.

[Reenskaug *et al.* 95]. T. Reenskaug, P. Wold, O.A. Lehne, *Working With Objects: The Ooram Software Engineering Method,* Prentice Hall, 1995.

[Schrefl et al. 96]. M. Schrefl, G. Kappel, P. Lang, ‘Modelling Cooperative Behaviour Using Cooperation Contracts,’ *Technical Report,* No. 3/96, Department of Information Systems, JKU Linz, 1996.

[Shaw *et al.* 95]. M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, ‘Abstractions for software architecture and tools to support them,’ *IEEE Transactions on Software Engineering,* April 1995.

[Soukup 95]. J. Soukup, ‘Implementing Patterns,’ in *Pattern Languages of Program Design,* J.O. Coplien, D.C. Schmidt (eds.), pp. 395-412, Addison-Wesley, 1995.

[Terry *et al.* 94]. A. Terry, F. Hayes-Roth, Erman, Coleman, Devito, ‘Overview of Teknowledge’s DSSA Program,’ *ACM SIGSOFT Software Engineering Notes,* October 1994.